CS 330 – Spring 2023 – Homework 07

- By: Gianna Sfrisi U77992006
- Collaborators: Albert Slepak U00956163

Problem 1. Marathon (10 points) You are training for the Boston Marathon by running various shorter races. There is one race every week. You have developed your own point system where for each race you award yourself a number of fitness points, based on how much you will gain from doing that race. Plan out which races to run so that you are as fit as possible by Marathon Day. As input you get the dates of the n races and the amount of points for each day in a table F. Design a dynamic programming algorithm for the below problems.

Note: Make sure to follow the steps we laid out for a full solution to a DP problem.

1. (Not to be handed in) Design your schedule if you can't run two weeks in a row. (It's fine to have a gap of more than one week.)

   As a note on why this problem is not so straightforward, consider the following example:

   $$F = [1, 10, 15, 10, 3]$$

   Here, it is not the case that you can greedily take the next available day, as this will have you selecting weeks $1, 3, and 5$, when the optimal is $2, 4$. Also, you cannot just simply select the largest remaining eligible value, as that would give you the same faulty response.

2. Design your schedule if you can run at most two weeks in a row and then have to rest a week. (It's fine to have gaps of more than a week.)

   Main Algorithm:

```
M = array(n + 1);
M[0] = 0;
for (int i = 1...n) do
    if i == 1 then
        M[1] = F[1];
    else
        if i == 2 then
            M[2] = F[2] + F[1]
        else
            c1 = M[i - 1];
            c2 = F[i] + M[i - 2];
            c3 = F[i] + F[i - 1] + M[i - 3];
            M[i] = max(c1, c2, c3);

return M[n];
```

Backtracking Algorithm:

```
if i == 0 then
    return $\emptyset$;
if i == 1 then
    return $\{f_1\}$;
if i == 2 then
    return $\{f_1, f_2\}$;
else
    c1 = M[i - 1];
    c2 = F[i] + M[i - 2];
    c3 = F[i] + F[i - 2] + M[i - 3];
    if c3 > c1 and c3 > c2 then
        return $\{f_i, f_{i-1}\} \cup$ Backtracking([$f_1$, $f_2$, ..., $f_n$], F, M, i - 3);
    if c2 > c1 and c2 > c3 then
        return $\{f_i\} \cup$ Backtracking([$f_1$, $f_2$, ..., $f_n$], F, M, i - 2);
    else
        Backtracking([$f_1$, $f_2$, ..., $f_n$], F, M, i - 1);
```

Proof of Accuracy for the Main Algorithm:

The algorithm uses a for loop to cycle through the list of possible races. In following the constraints that there cannot be more than two consecutive races completed, there are three different cases of possible combinations to consider. The algorithm checks the base cases if the race is the first or second in the list. For the first race in the list, the amount of fitness points are initialized; for the second index in the race, the first two race's points are summed together. The first case to consider is if the current race is not run. The solution, in this case, would be the maximum number of fitness points for the prior race. The second case is if the current race is run. The amount of fitness points for the current race is added to the optimal solution for race two prior. If the current race's points were added to the fitness points of the race directly beforehand, the algorithm could not guarantee that there would not be more than two consecutive races, hence if the current race is added the race prior must be skipped. The third case adds the points of the current race, and the race prior; however, instead of taking the optimal total points of the next race in the list, it skips and takes the optimal amount of points of the race's three indexes away. This guarantees that there are not more than two races back to back. The maximum total amount of fitness points between the three cases is returned.

Proof of Accuracy for the Backtracking Algorithm:
The goal of the backtracking algorithm is to return the path required to maximize the total amount of fitness points. Using the base cases where the index equals 0, 1, or 2: the algorithm returns the value of 0 at index 0 since there is no race currently in the list, for the index of 1 the algorithm returns the index itself, and for the index of 2, it returns the two races in the list. Since the maximum length in the base cases is 2, it is guaranteed there will not be more than two consecutive races. However, when the length of the list is greater than two, the algorithm needs to check the number of points at each index given the constraints of the problem and return the indexes associated with that path with the resulting optimal solution. The backtracking algorithm uses the same cases as the main algorithm which are outlined above. However, instead of immediately picking the maximum value of the three cases, the backtracking algorithm compares the relationship between the cases and returns the lists accordingly.

Run-time Analysis for the Main Algorithm:
The algorithm runs at a total of $O(n)$ time. The for loop cycles through each race to find the combination of races with the optimal number of points. Since the algorithm uses the dynamic programming method and references the optimal point value for the previous races, the for loop will only cycle through the list of races once. Additionally, each of the calls in the if and else blocks would run at time $O(1)$ since it is a single arithmetic calculation. Therefore the total run time would be $O(n)$.

Run-time Analysis for the Backtracking Algorithm:
Similar to the main algorithm, the run-time of the backtracking algorithm is $O(n)$. The backtracking algorithm has a series of if statements that all run at $O(1)$ time. The backtracking algorithm is called accordingly based on the size relationship between the three cases. Since the algorithm is called for each possible race, it will be called $O(n)$ times.

3. Design your schedule if you can run at most m weeks in a row and then have to rest a week. (It's fine to have gaps of more than a week.) For this part only you may leave out the backtracking algorithm for full credit.

   Main Algorithm:

```
M = array(n + 1);
M[0] = 0;

for (int j = 1...n) do
    if j < m then
        M[j] = sum(F[1], F[2], ..., F[j])
    else
        for i = 0...m do
            $c_i$ = F[j] + F[j - 1] + ... + F[j - i] + M[j - i - 1]
        M[j] = max($c_0$, $c_1$, ..., $c_m$);

return M[n];
```

   Proof of Accuracy for the Main Algorithm:
   The algorithm uses the same ideas as the previous main algorithm but is more generalized to fit any constraint amount of consecutive weeks. In the case where the index is less than the total amount of consecutive weeks, the amount of fitness points for every race is summed together. Since there are fewer weeks than the constraint, the algorithm does not have to pick and choose races based on their points; every race is able to be added. In the case where there are more races than the constraint, a for loop is used to find the optimal combination of races by finding each possible combination and taking the maximum.

   Run-time Analysis for the Main Algorithm:
   The algorithm runs at time $O(mn)$. The algorithm loops through the total number of races in the list, taking into account the different possibilities given the consecutive week constraint. Given the list of total races n, each of the combinations of weeks is explored therefore the total run-time is $O(mn)$, n representing the number of races, and m representing the amount of allowed consecutive weeks.

Problem 2. Pirate loot (10 points) Two pirates are trying to divide their loot in a way that they both find fair. They randomly lay out the treasure in a line, each of the n items has a certain value v_i that is given as part of the input. The pirates take turns where they can take the item either at the left or right end of the line. Design an algorithm for the first pirate to collect as much value as possible. You may assume that both pirates use the optimal strategy.

Main Algorithm:

```
M = array(n x n);
for i = 1...n do
    for j = 1...n do
        if i == j
            M[i][j] = $v_i$;
        else
            M[i][j] = 0;
for i = 1...n do
    for j = n...i + 1 do
        if i == j
            M[i][j] = $v_i$;
        else
            leftPath = $v_i$ + min(M[i + 2][j], M[i + 1][j - 1]);
            rightPath = $v_j$ + min(M[i + 1][j - 1], M[i][j - 2]);
            M[i][j] = max(leftPath, rightPath);
return M[1][n];
```

Backtracking Algorithm:

```
if i == j then
    // Base case where only one item is left
    treasureIndices.append(i);
    return treasureIndices;
if M[i][j] - $v_i$ == M[i + 2][j] then
    treasureIndices.append(i);
    return Backtracking([$v_1$, $v_2$, ..., $v_n$], M, treasureIndices, i + 2, j);
if M[i][j] - $v_i$ == M[i + 1][j - 1] then
    treasureIndices.append(i);
    return Backtracking([$v_1$, $v_2$, ..., $v_n$], M, treasureIndices, i + 1, j - 1);
if M[i][j] - $v_j$ == M[i + 1][j - 1] then
    treasureIndices.append(j);
    return Backtracking([$v_1$, $v_2$, ..., $v_n$], M, treasureIndices, i + 1, j - 1);
if M[i][j] - $v_j$ == M[i][j - 2] then
    treasureIndices.append(j);
    return Backtracking([$v_1$, $v_2$, ..., $v_n$], M, treasureIndices, i, j - 2);
```

Proof of Accuracy for the Main Algorithm:
The main algorithm has two key components each iterated through a nested for loop. The first set of for loops assigns the weight for every pair of indices representing the line of treasure. If an index exists, it is assigned the cost of that treasure item. If the index does not exist it is assigned a value of 0. The second set of nested for loops checks which path would be the most beneficial for the pirate whose turn it is. The base case checks if the index is pointing to the last piece of treasure in the list, which the pirate whose turn it is would have to take without choice. The else statement then calculates the total gain of choosing the treasure on the left side of the line and on the right side. The weight and values left over for the next pirate are both taken into account and the maximum path is the one stored.

Proof of Accuracy for the Backtracking Algorithm:
The backtracking algorithm starts by taking into account the base case where i and j are equal to each other, hence there is one piece of treasure left that must be chosen next. The remainder of the algorithm consists of a series of if statements used to determine the path. The first and fourth if statements check the impact on the total overall weight of taking the last element on either the left or right side. The second and third if statements take more of a symmetric approach to check how this choice will impact the choice of the next pirate. The most optimal path out of the four options is then returned and the pattern continues.

Run-time Analysis for the Main Algorithm:
The algorithm is consisting two different sets of nested for loops. The initial set of for loops set the value at each index whereas the second set finds the optimal path. Since the outer loop iterates through 1 to n, the overall run-time for that loop would be $O(n)$. Similarly, the inner loop also cycles through 1 to n. The total run-time for the nested for loop would be $O(n^2)$ since it will iterate through each pair of values for i and j. The second set of nested for loops also cycle through 1 to n and would also have a total run-time of $O(n^2)$. In combining the two algorithms the total run-time would be $O(2n^2)$ which would simply to $O(n^2)$

Run-time Analysis for the Backtracking Algorithm:
The total run-time of the backtracking algorithm is $O(1)$. Since each if statement runs at time $O(1)$ and the entire algorithm is a series of 5 if statements, the total run-time would be $O(1) + O(1) + O(1) + O(1) + O(1)$ which simplifies to just $O(1)$.